2

User's Guide
CMU/SEI-89-UG-6
ESD-TR-89-12

Carnegie-Mellon University
Software Engineering Institute

SEI Serpent Application Developer's Guide

January 1989

User's Guide
CMU/SEI-89-UG-6
ESD-89-TR-12
January 1989

SEI Serpent Application Developer's Guide

User Interface Prototyping Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This user's guide was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this guide should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This guide has been reviewed and is approved for publication.

FOR THE COMMANDER

[illegible]
Karl Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

SEI Serpent Application Developer's Guide

1. Introduction

Serpent is a User Interface Management System (UIMS) that supports the development and execution of a user interface of a software system. Serpent supports incremental development of the user interface from the prototyping phase through production to maintenance or sustaining engineering. Serpent encourages a separation of functionality between the user interface portion of an application and the functional portion of an application. Serpent is also easily extended to support additional input/output technologies.

This **Application Developer's Guide** describes how to develop applications using Serpent. The contents of this guide assume that you have read and understood the concepts described in "An Introduction to Serpent"; it also assumes that you are experienced with using C or Ada. (KR)

Parts

This guide's major parts are:

- **Overview:** The overview part of this guide provides a general description of the role of an application in a software system developed using Serpent. It also describes a conceptual framework for application development.
- **C Language Application Development:** The two sections in this part address the needs of C language application developers. The first section describes how to develop an application using Serpent in the C programming language. The second section contains a complete set of descriptions of all the constants, types, and routines available to the C language Serpent application developer.
- **Ada Language Application Development:** The two sections in this part address the needs of Ada language application developers. The first section describes how to develop an application using Serpent in the Ada programming language. The second section contains a complete set of descriptions of all the constants, types, and routines available to the Ada language Serpent application developer.
- **Application and Dialogue Testing** The two sections in this part of this guide describe an approach for testing the application and dialogue portions of a software system developed using Serpent, independent of the implementation language selected. The first section describes the task steps involved in testing while the second section describes commands available to the application or dialogue tester.

The glossary provides definitions and explanations of terms that are used in this guide.

References

The purpose of this guide is to provide you with sufficient information to develop Serpent applications. The following publications address other aspects of Serpent.

- **An Introduction to Serpent:** Describes the roles involved in the use of Serpent, the components of Serpent, and the steps involved in using Serpent for particular applications.
- **SADDLE User's Guide:** Explains how to use the SADDLE language and preprocessor.
- **Dialogue Editor User's Guide** Describes how to develop and maintain a dialogue using the dialogue editor.
- **Slang Reference Manual:** A complete reference to the Slang dialogue specification language.
- **Guide to Adding Technologies:** Describes how to add I/O technologies to Serpent or an existing Serpent application.

2. Overview

A main goal of Serpent is to encourage the separation of a software system into an application portion and an user interface portion in order to provide the application developer with a *presentation independent* interface. The *application* portion consists of those components of a software system that implement the "core" application functionality of a system. The *user interface* portion consists of those components that implement an end-user *dialogue*. A dialogue is a specification of the presentation of application information and end-user interactions.

During the design stage, the system designer decides which functions belong in the application component and which belong in the user interface component of the system.

2.1. The Serpent Architecture

Serpent is implemented using a standard UIMS architecture. This architecture (see figure 2-1) consists of three major layers: the *presentation layer*, the *dialogue layer*, and the *application layer*. The three different layers of the standard architecture are viewed as providing differing levels of end-user feedback.

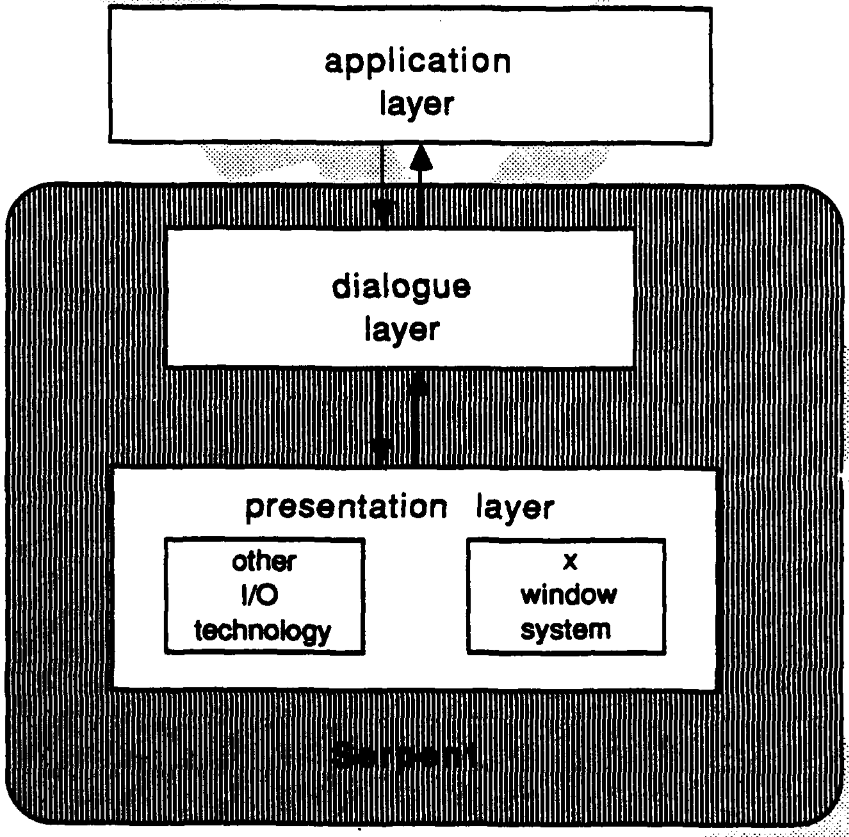


Figure 2-1: Serpent Architecture

The presentation layer consists of various input/output technologies which have been incorporated into Serpent. Input/output technologies are existing hardware/software systems that perform some level of generalized interaction with the end-user. Serpent is being distributed with an interface to the X Window System, version 11. Other input/output technologies can be integrated with Serpent. See the *Guide to Adding Technologies* for a discussion of how this can be accomplished.

One way of viewing the three levels of the architecture is the level of functionality provided for user input. The presentation layer is responsible for lexical functionality, the dialogue layer for syntactic functionality, and the application layer for semantic functionality. In terms of a menu example, the presentation layer has responsibility for determining which menu item was selected and for presenting feedback which indicates which choice is currently selected. The dialogue layer has responsibility for deciding whether another menu is presented and presenting it or whether the choice requires application action. The application layer is responsible for implementing the command implied by the menu selection.

The end-user interface for a software system is specified formally as a *dialogue*. The dialogue is executed by the *dialogue manager* at runtime in order to provide a end-user interface for a software system. The dialogue specifies both the presentation of application information and end-user interactions. The Serpent dialogue specification language (SLANG) allows dialogues to be arbitrarily complex.

The application provides the functional portion of the software system in a presentation independent manner. It may be developed in C, Ada, or other programming languages. The application may be either a functional simulation for prototyping purposes or the actual application in a delivered system. The actions of the application layer are based on knowledge of the specific problem domain.

2.2. Shared Database

Serpent provides an active database model for specifying the user interface portion of a system. In an active database, multiple processes are allowed to update a database. Changes to the database are then propagated to each user of the database. This active database model is implemented in Serpent by a *shared database* that logically exists between the application and I/O technologies. The application can add, modify, query, or remove data from the shared database. Information provided to Serpent by the application is available for presentation to the end-user. The application has no knowledge of the presentation media or user interface styles used to present this information.

Information in the shared database may be updated by either the application or I/O technologies. Figure 2-2 illustrates the use of the shared database in Serpent.

Serpent allows the specification of dependencies between elements in the shared database in the dialogue. These dependencies define a mapping between application data, presentation objects and end-user input. The dialogue manager enforces these dependencies by

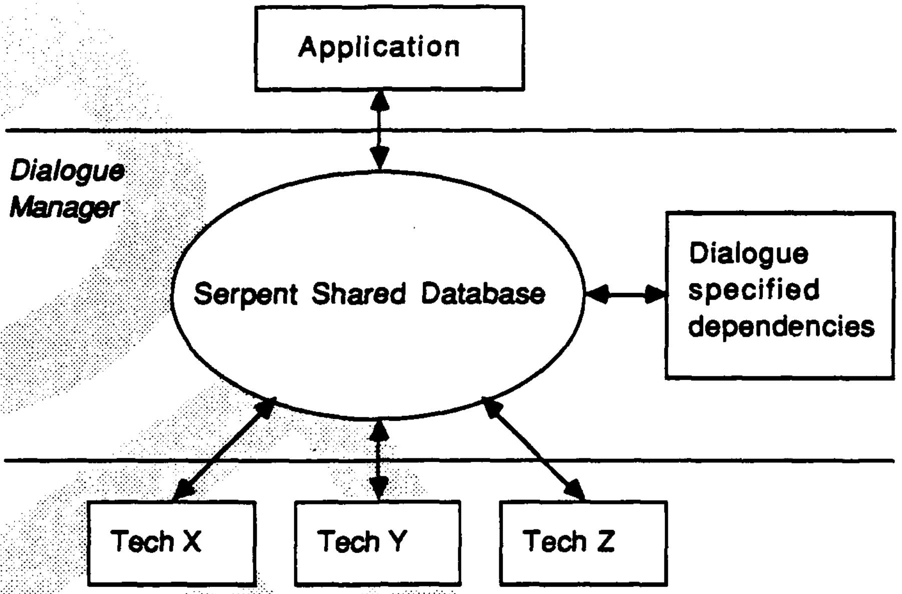


Figure 2-2: Shared Database

operating on the information stored in the shared database until the dependencies are met. Changes are then propagated to either the application or the I/O technologies as appropriate. See the *SLANG Reference Guide* for a further discussion.

The *type* and *structure* of information that can be maintained in the shared database is defined externally in a *shared data definition* file. This corresponds to the database concept of *schemas*. A shared data definition file is required for each application.

A shared data definition file consists of both aggregate and scalar data structures. Top-level data structures become *shared data elements* that may be instantiated at runtime. Nested data structures become components that are considered part of the shared data element. Serpent does not allow nesting of records.

It is possible to define multiple instances of a single shared data element. Shared data elements are instantiated by specifying the element name. Each *shared data instance* is identified by a unique *ID*. IDs must be maintained by the application to identify shared data instances when multiple instances of a single shared data element exist. Figure 2-3 provides an illustration of shared data instantiation.

Serpent supports both a synchronous and asynchronous system model. This is necessary since an application often needs to satisfy real-time constraints and cannot necessarily afford to wait for end-user input. This introduces a situation where multiple processes, which are using the shared database, may access or modify the database concurrently. This concurrent access of the shared database may result in a situation where the *integrity* of the database is corrupted.

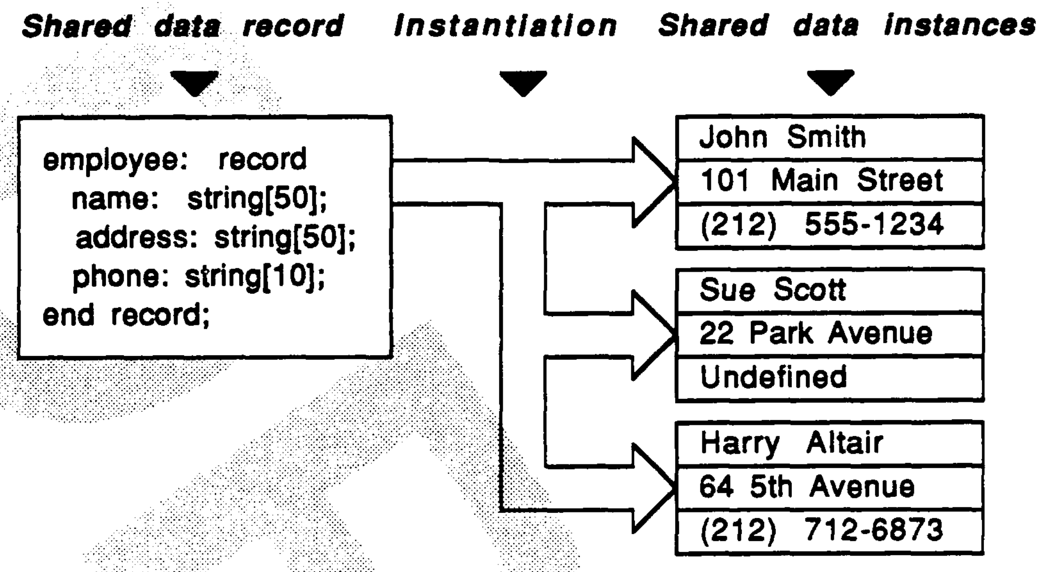


Figure 2-3: Shared Data Instantiation

This problem is solved in Serpent through the use of database concurrency control techniques. Updates to the Serpent shared database are packaged in *transactions*. Transactions are collections of updates to the shared database that are logically processed at one time. Transactions can be *started*, *committed*, or *rolled back*. Committing a transaction causes the updates to be made to the shared database. Rolling back a transaction causes termination of the transaction. A transaction which is started but not yet either committed or rolled back is said to be *open*. There may be several transactions open at the same time.

2.3. Application Development

There are three major tasks which need to be performed when developing an application for Serpent:

1. Define shared data.
2. Add information to shared data.
3. Retrieve information from shared data.

To perform each of the preceding tasks, there are several steps you need to complete. The first task is completely independently of the language in which the application was developed. The last two tasks are also both language independent, but how you specify them depends on the programming language you chose for application development. Currently Serpent supports two different language interfaces, C and Ada. Therefore, the two parts that follow specify and illustrate how to develop an application in the C and Ada programming languages, respectively.

2.4. Application and User Interface Testing

The record/playback feature of Serpent allows you to record transactions between the application and dialogue manager, or between the dialogue manager and the various technologies. These transactions may then be played back at a later time. This is useful in performing regression and/or stress testing of the application, dialogue or technologies.

There are two major tasks that need to be performed when using the record/playback feature of Serpent for testing the application or user interface:

1. Recording shared database transactions.
2. Testing the application or user interface.

To perform each of the preceding tasks, there are several steps you need to complete. The specification of steps in the first task is dependent on the programming language selected for application development. Therefore, a description of the steps involved in recording shared database transactions is included in both the C language and Ada language application development parts of this guide.

The execution of the steps in the second task are performed independently of the language in which the application was developed. These tasks are described in the Application and Dialogue part of this guide.

2.5. Sensor Site Status Example

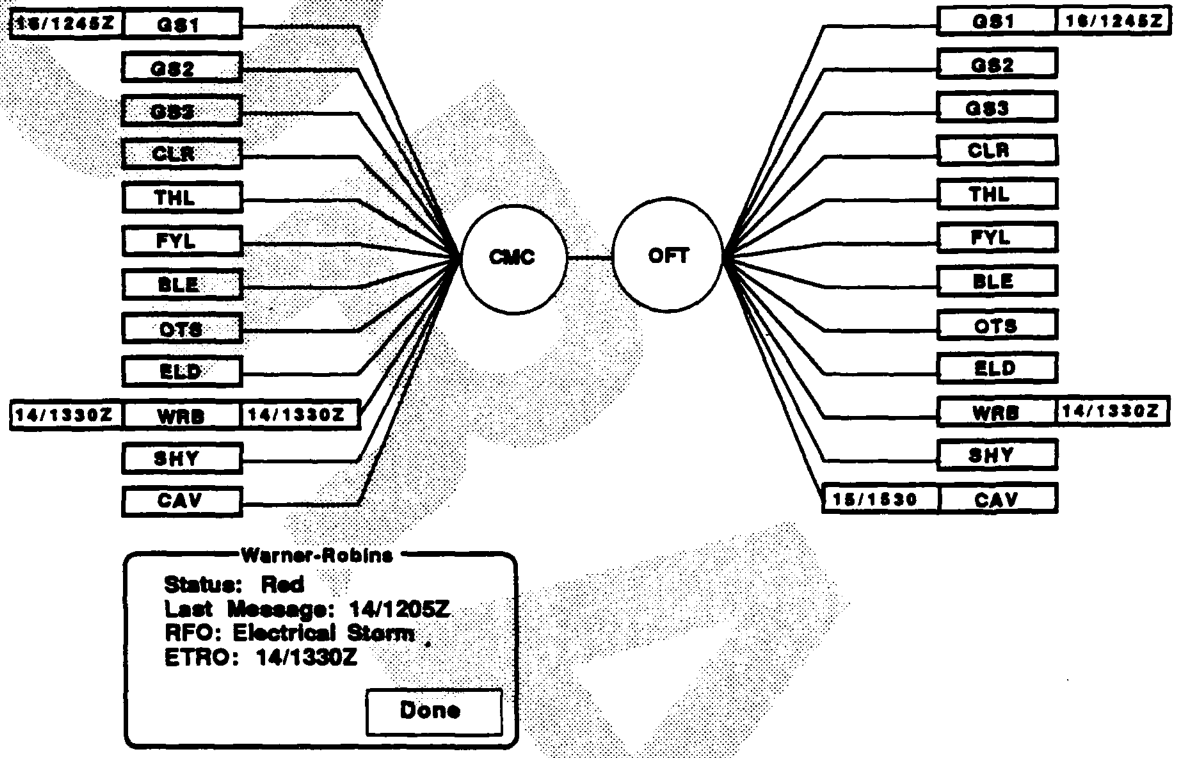
The sensor site status (SSS) application is an example of an application developed using Serpent. Figure 2-4 is an illustration of the "spider chart" display which is one possible end-user interface for the application.

The sensor site status application is adapted from a command and control application. The purpose of the application is to monitor and display the status of various sensor sites and their associated communications lines to the two correlation centers.

The columns of rectangular boxes on the right and left sides of the spider chart display (for example, GS1, GS2) represent sensor sites. The circles in the middle of the display represent the correlation centers that collect information from the sensors. Each sensor site communicates with both correlation centers; this is represented by the duplication of sensor site boxes on both the right and left sides of the display. The lines present the communication lines between the sensor sites and the correlation centers.

An operator may display detailed information concerning a sensor site by selecting a sensor site box corresponding to that sensor. This causes a detailed window to appear displaying information concerning the status of the sensor, the date and time of the last message, the reason for outage (RFO) and the estimate time to returned operation (ETRO). These fields may be modified by the operator. Sensors may have one of three status: green, yellow and red. For sensor which are not fully operational (i.e. status is green) the ETRO is displayed to the outside of the sensor site box. ETROSs are also displayed over communication lines that are not fully operational.

Throughout this guide, a reference to the sensor site status application implies a reference to this example.



Warner-Robins
Status: Red
Last Message: 14/1205Z
RFO: Electrical Storm
ETRO: 14/1330Z

Figure 2-4: Sensor Site Display

3. C Language Application Development

This part includes two sections:

- How to Develop an Application in C: A step by step specification of the tasks involved in developing a Serpent application in the C programming language.
- Serpent C Language Interface Reference: A detailed description of the types, constants and routines available for developing Serpent applications in the C programming language.

3.1. How to Develop an Application in C

The main tasks for developing an application for Serpent require that you define the shared data, add information to shared data, and retrieve information from shared data. There are also two additional tasks which may applied: recording and checking status. Each of these tasks is described in the subsections that follow.

3.1.1. Task 1: Defining the Shared Data

Defining shared data involves two steps:

1. Create the shared data definition file.
2. Run the created file through the SADDLE processor.

The following is a brief description of each of these two steps. The *SEI Serpent SADDLE User's Guide* contains a more complete description of both these steps.

Step 1: Create the shared data definition file The shared data definition file defines the type and structure of application information that may be maintained by the Serpent shared database. The shared data definition is defined in an external ASCII file in SADDLE.

Figure 3-1 is a example of a shared data definition file for the sensor site status application. The content of the shared data definition file is independent of the implementation language used.

The file shown in Figure 3-1 contains definitions for the data shared between the application and the dialogue for the sensor site status application. The three records define the type and structure of the sensor, correlation center, and communication line application objects. Note that these records only contain information to define the actual objects; they do not specify how the information is presented to the end user.

Step 2: Run the created file through the SADDLE processor. Once the shared data has been defined, you can run the file through the SADDLE processor to generate a C language header file. You then include this header file with your C application in order to declare local variables of the shared data types. This allows you to directly manipulate shared data structures in C. The C header file generated by running the shared data definition file shown in Figure 3-1 through the SADDLE processor is illustrated in Figure 3-2.

```
sensor_site_status:shared data

sensor:  record
  site_abbr:string[3];
  status:integer;
  site:string[50];
  last_message: string[8];
  rfo:string[50];
  etro:string[8];
end record;

correlation_center:record
  name:string[3];
  status:  integer;
end record;

communication_line:record
  from_sensor:id of sensor;
  to_cc:id of correlation_center;
  etro:string[8];
  status:integer;
end record;

end shared data;
```

Figure 3-1: A shared data definition file

```
#define MAIL_BOX "sss_mailbox"
#define ILL_FILE "sss.ill"

typdef struct   {
  char site_abbr[4]
  int  status;
  char site[51];
  char last_message[9];
  char rfo[51];
  char etro[9];
  }    sensor;

typedef struct  {
  char name[4];
  int  status;
  }    correlation_center;

typedef struct  {
  id_type    from_sensor;  /*ID of sensor */
  id_type    to_cc;        /*ID of correlation_center]*/
  char etro[9];
  int  status;
  }    communication_line;
```

Figure 3-2: C language header file

In Figure 3-2, the first two lines in the file define two well-known constants: `MAIL_BOX` and `ILL_FILE`. These constants are used in task 2 to initialize Serpent. The three **typedefs** correspond to the records defined within the shared data definition file.

3.1.2. Task 2: Adding Information to the Shared Database.

Once you have defined the application shared data, you can begin to develop the application. The code segment from the sensor site status application in Figure 3-3 illustrates the basic operations for adding information to the shared database.

```
#include "serpent.h"   /* serpent interface definition */
#include "sss.h"       /* application data structures  */

#define GREEN_STATUS 0
#define YELLOW_STATUS 1
#define RED_STATUS 2

main()
{
  transaction_type transaction; /* transaction handle  */
  correlation_center cmc,oft;   /* correlation centers */

  serpent_init(MAIL_BOX,ILL_FILE);

  strcpy(oft.name, "OFT");
  oft_status = GREEN_STATUS;

  transaction = start_transaction();

  cmc_id = add_shared_data(
    transaction,
    "correlation_center",
    NULL,
    &cmc
  );

  oft_id = add_shared_data(
    transaction,
    "correlation_center",
    NULL,
    &oft
  );

  commit_transaction(transaction);

  serpent_cleanup();
  return;
}
```

Figure 3-3: Basic calls for adding information to the shared database

Preliminary Task Steps

In preparation for the task of adding information, you need to complete two preliminary steps:

1. Include header files.
2. Define local variables.

Step P1: Include header files. Include the two header files, as shown in Figure 3-3. The first of these two files is called **serpent.h** and contains the definition for the Serpent external interface. This must be included first since it contains type definitions that are used in the second file. The second file that needs to be included is the C language header file generated in the previous step, when you ran the shared data definition through the SADDLE processor. This file, **sss.h** in the example, defines the structure of the shared data.

Step P2: Define local variables. The next preliminary step is to define the required local variables. The first variable defined is transaction, which is of **`transaction_type`**. This variable maintains the handle for a created transaction. The next variables to be defined are **`cmc`** and **`oft`**, both of which are of type **`correlation_center`**. These variables store local instances of the data that is going to be shared across the interface with the Serpent system. The type definition for the **`correlation_center`** structure was automatically generated by the SADDLE processor during step 2 of task 1.

The two variables that follow, **`cmc_id`** and **`oft_id`**, store the ids of the shared data instances created in shared data. It is necessary for the application to maintain this information, since it is the only way to correlate end-user updates with local application information when multiple instances of a single shared data element are used.

Main Task Steps

The main task of adding information to the shared database involves five distinct coding steps:

1. Initialize Serpent.
2. Start a transaction.
3. Add shared data to the interface.
4. Commit the transaction.
5. Clean up.

Step 1: Initialize Serpent. Once the appropriate variables have been declared it is possible to begin describing the logic. The first step is to initialize the Serpent system using the **`serpent_init`** call and passing the **`MAIL_BOX`** and **`ILL_FILE`** constants generated by the SADDLE processor during step 2 of task 1.

Step 2: Start a transaction. Before information can be added to the shared database it is necessary to start a transaction. All additions or modifications to the shared database must be performed as part of a transaction.

Step 3: Add information to the shared database. Once a transaction has been started, you can begin to add information to the shared database as part of this transaction.

Step 4: Commit the transaction. The actual change to shared data does not occur until the transaction is committed. Up to this point it is also possible to roll back the transaction so that none of the changes to shared data occur.

Step 5: Clean up. The `serpent_cleanup` routine must be invoked before exiting the application. It is important that you complete this step, to release all allocated system resources.

3.1.3. Task 3: Retrieving Information from the Shared Database

Once application data exists in the shared database it may be presented to the end-user using one or more of the available technologies. The end-user may in turn make modifications to this data. These modifications are sent back to the application to be updated in the application's local database. It is therefore necessary for the application to retrieve information back from the shared database.

The Serpent interface provides both synchronous and asynchronous calls for getting information back from the shared database. The following code segment from the sensor site status application in Figure 3-4 illustrates the basic calls required to synchronously retrieve data from the interface.

```
{
/*
   Retrieve information from shared database.
*/
  transaction = get_transaction();

  id = get_first_changed_element(transaction);

  while (id != null_id) {

    shared_data_element = get_from_hashtable(id_table, id);
    incorporate_changes(transaction,id,shared_data_element);

    id = get_next_changed_element(transaction);
  }

  purge_transaction(transaction);

}
```

Figure 3-4: Basic calls required to retrieve data synchronously

Task Steps

The task of retrieving information from the shared database involves three distinct coding steps:

1. Get a transaction.
2. Update local database.
3. Purge transaction.

Step 1: Get a transaction. The first step in retrieving information from the shared data base is to get a transaction. The `get_transaction` routine waits until a transaction is available and then returns a handle for this transaction. To poll for a new transaction asynchronously, it is possible to call the `get_transaction_no_wait` routine, which will return `not_available` if no transaction is available.

Step 2: Update local database. Transactions can be thought of logically as a list of changed elements. The next call, `get_first_changed_element`, returns the id of the first changed element on the list. This id can then be used to access several types of information about the shared data element.

The application must maintain a correlation between the shared data ids and the actual data items to incorporate changes successfully into its existing local data. For the purposes of this example, it is assumed that this database is maintained as a hashtable indexed by the shared data element id. The purpose of the while loop then is to incorporate all of the changes into this local database or hashtable. A pointer to the shared data element to be updated is retrieved from the hashtable using the `get_from_hashtable` routine and passing id as an index into the hashtable. The `incorporate_changes` call then makes the updates to the local description of the shared data elements and whatever changes were made by the dialogue.

The last call within the loop gets the next changed element from the transaction. The loop repeats until a `null_id` is returned.

Step 3: Purge transaction. After the loop ends the transaction can be purged safely. It is you. responsibility to ensure that transactions are purged, since this call releases resources that otherwise could run out.

3.2. Recording Shared Database Transactions

There are two major tasks that need to be performed when using the record/playback feature of Serpent for testing the application or user interface:

1. Recording shared database transactions.
2. Testing the application or user interface.

Since the steps involved in the second task can be performed independently of the implementation language they are described later in the Application and Dialogue Testing part of this guide.

Before testing the application or the dialogue however, you must first record the transactions you would like to use in testing. Figure 3-5 illustrates the basic operations for recording transactions.

```
{
  transaction_type transaction;

                    :
/*
   Start recording.
*/
   start_recording("recording", "test data:  5.7.3");
/*
   Send test data.
*/
   transaction = start_transaction();

                    :

   commit_transaction(transaction);

   transaction = start_transaction();

                    :

   commit_transaction(transaction);

   transaction = start_transaction();

                    :

   commit_transaction(transaction);
/*
   Stop recording.
*/
   stop_recording();

                    :
}
```

Figure 3-5: Recording transactions

Task Steps

There are three distinct coding steps involved in recording shared database transactions:

1. Start recording.
2. Send transactions.
3. Stop recording.

Step 1: Start recording. The first step is to begin recording by calling the `start_recording` routine specifying both the name of the file in which to save the recording and a message to help identify the file.

Step 2: Send transactions. After the call is made you may start sending transactions across the interface. You may send any number of transactions containing any type or amount of data.

Step 3: Stop recording. Once `start_recording` has been called, all transactions and associated data will be saved out to the specified file until the `stop_recording` routine is invoked.

3.2.1. Checking Status

Each routine in Serpent sets status on exit. It is good software engineering practice to check this status after every call to make sure that the routine has executed correctly, and provide appropriate recovery actions if it has not. Figure 3-6 shows the operations that Serpent provides for examining the status.

```
transaction = start_transaction();
if(get_status()) {
  print_status("error during start_transaction");
  return;
}
```

Figure 3-6: Operations for examining the status

The first of these status calls is the `get_status`, which returns an enumeration of status codes. Valid status that each routine in Serpent may return are defined in the reference sections of this developer's guide. Successful execution (or "OK") is always set to zero; hence, it is possible to make the simple boolean comparison shown in Figure 3-6 for bad status.

The `print_status` routine prints out a user-defined error message and the current status.

3.3. Serpent C Language Interface

3.3.1. Types and Constants

This subsection contains the type and constant definitions that are used in the C language interface to the Serpent system. The following is a list and short description of each of these types and constants. A more complete description immediately follows:

Type/Constant	Description
`buffer`	used to define the structure of a shared data buffer
`change_type`	defines the type of modification made for an element
`id_type`	used to uniquely identify shared data elements
`null_id`	defines the null value for the `id_type`
`serpent_data_types`	an enumeration of defined Serpent data types
`transaction_type`	used to define transaction handles
`undefined values`	Constants corresponding to undefined values for all supported types

TYPE

buffer

DESCRIPTION The **buffer** type is used to define the structure of a buffer within shared data.

DEFINITION

```
typedef struct {
  int length;
  caddr_t body;
} buffer;
```

COMPONENTS

length	Length of the buffer in bytes.
body	Address of the actual buffer data.

change_type

DESCRIPTION The change_type defines the type of modification made for an element.

DEFINITION

```
typedef enum change_type {
  no_change   = -1,
  create = 0,
  modify = 1,
  remove = 2,
  get = 3
} change_type;
```

COMPONENTS

no_change	Not changed or invalid change.
remove	Remove existing shared data instance.
create	Newly created shared data instance.
modify	Modified existing shared data instance.
remove	Remove existing shared data instance.
get	Get value for existing shared data instance.

TYPE

id_type

DESCRIPTION	The id type is used to uniquely identify shared data elements.
DEFINITION	`typedef private id_type;`

null_id

DESCRIPTION

The null_id constant defines the null value for the id_type. This constant can be used to test for null id values.

DEFINITION

```
#define null_id (iid_id_type)-1
```

serpent_data_types

DESCRIPTION

The serpent data types type is an enumeration of the defined Serpent data types.

DEFINITION

```
typedef enum data_types {
  serpent_null_data_type =-1,
  serpent_boolean =0,
  serpent_integer =1,
  serpent_real =2,
  serpent_string =3,
  serpent_record =4,
  serpent_id =5,
  serpent_buffer =6,
  serpent_undefined =7
} serpent_data_types;
```

TYPE

transaction_type

DESCRIPTION	Variables of transaction_type are used to define transactions.
DEFINITION	`typedef private transaction_type;`

undefined values

DESCRIPTION

The following constants correspond to undefined values for all types supported by Serpent. These constants can be used to test for ndefined shared data components. When checking for an undefined record value it is best to check the buffer length failed for **UNDEFINED_BUFFER_LENGTH**.

DEFINITION

```
#define UNDEFINED_BOOLEAN (boolean)0xAAAAAAAA
#define UNDEFINED_INTEGER (int)0xAAAAAAAA
#define UNDEFINED_REAL (double)0xAAAAAAAAAAAAAAAA
#define UNDEFINED_STRING (string)0xAAAAAA00
#define UNDEFINED_RECORD (caddr_t)0xAAAAAAAA
#define UNDEFINED_ID (iid_id_type)0xAAAAAAAA
#define UNDEFINED_BUFFER_LENGTH (int)-1
#define UNDEFINED_BUFFER_BODY (caddr_t)0xAAAAAAAA
```

3.3.2. Routines

This subsection describes the routines that make up the C language interface to Serpent. These routines fall into the following categories:

- Initialization/cleanup
  - serpent_init
  - serpent_cleanup
- Transaction processing
  - start_transaction
  - commit_transaction
  - rollback_transaction
  - get_transaction
  - get_transaction_no_wait
  - purge_transaction
  -
- Sending and retrieving data
  - add_shared_data
  - put_shared_data
  - remove_shared_data
  - get_first_changed_element
  - get_next_changed_element
  - get_shared_data
  - incorporate_changes
  - create_changed_component_list
  - destroy_changed_component_list
  - get_change_type
  - get_element_name
  - get_shared_data_type
  -
- Record/playback
  - start_recording
  - stop_recording
- Checking Status
  - get_status
  - print_status

add_shared_data

DESCRIPTION

The **add_shared_data** routine creates an instances for the specified shared data element and returns a unique ID. The shared data instanced may or may not be initialized.

SYNTAX

```
id_type add_shared_data(
  /* transaction : in transaction_type */
  /* element_name : in string */
  /* component_name : in string */
  /* data : in caddr_t */
);
```

PARAMETERS

transaction	The transaction for which this operation is defined.
element_name	The name of the shared data element.
component_name	The name of a specific component to be initialized with the data or null if the data corresponds to the entire element.
data	data or null pointer if non-initialized.

RETURNS

The ID of the newly created shared data instance.

STATUS

ok, out_of_memory, null_element_name, overflow)

commit_transaction

DESCRIPTION	The `commit_transaction` procedure is used to commit a transaction to the shared database.

SYNTAX

```
void commit_transaction(
  /* transaction: in transaction_type */
);
```

PARAMETERS

`transaction`	Existing transaction ID.

STATUS `ok, out_of_memory, invalid_transaction_handle`

create_changed_component_list

DESCRIPTION	The **create_changed_component_list** function accepts an instance id as a parameter and creates a list of changed component names.

SYNTAX

```
LIST create_changed_component_list(
   /* id: in id_type */
);
```

PARAMETERS	**id**	Existing data instance id

RETURNS	The list of changed component names associated with a data instance, or **NULL** if none.
STATUS	**ok, invalid_id, out_of_memory, element_not_a_record**

destroy_changed_component_list

DESCRIPTION

The **destroy_changed_component_list** procedure releases storage associated with a changed component list.

SYNTAX

```
void destroy_changed_component_list(
   /* changed_component_list : in out LIST */
);
```

PARAMETERS

changed_component_list
List to be destroyed.

STATUS

ok

get_change_type

DESCRIPTION

The `get_change_type` function accepts an instance id as a parameter and returns the associated change type.

SYNTAX

```
change_type get_change_type(
  /* is : in id_type */
);
```

PARAMETERS

`id`	Existing shared data ID

RETURNS

Element name associated with the shared data instance ID.

STATUS

`ok`, `invalid_change_type`, `invalid_transaction_handle`, `invalid_id`

get_element_name

DESCRIPTION	The **get_element_name** function accepts an instance id as a parameter and returns the associated element name.	
SYNTAX	`string get_element_name( /* id : in id_type */ );`	
PARAMETERS	**id**	Existing shared data ID.
RETURNS	Element name associated with the shared data instance ID	
STATUS	**ok, invalid_id**	

SYNTAX:

```
string get_element_name(
  /* id : in id_type */
);
```

FUNCTION

get_first_changed_element

DESCRIPTION

The **get_first_changed_element** function is used to get the id of the first changed element on a transaction list.

SYNTAX

```
id_type get_first_changed_element(
  /* transaction_type : in transaction */
);
```

PARAMETERS

transaction	Existing transaction ID

RETURNS

The handle of the first changed element

STATUS

ok, invalid_transaction_handle, out_of_memory

FUNCTION

get_next_changed_element

DESCRIPTION	The **get_next_changed_element** function is used to get the id of the next changed element on a transaction list or return **null_id** if the transaction list is empty.

SYNTAX

```
id_type get_next_changed_element(
  /* transaction_type : in transaction */
);
```

PARAMETERS		
	transaction	Existing transaction ID

RETURNS	The handle of the next changed element
STATUS	**ok, invalid_transaction_handle, out_of_memory**

get_shared_data

DESCRIPTION The **get_shared_data** function allocates process memory, copies shared data into process memory and returns a pointer to the data.

Warning: Record components may not have been specified and, therefore, would not contain valid data.

SYNTAX

```
caddr_t get_shared_data(
  /* transaction : in transaction_type */
  /* id : in id_type */
  /* component_name : in string */
);
```

PARAMETERS

transaction	Transaction in which to find the shared data id.
id	Existing shared data id.
component_name	Name of component for which to retrieve data, or entire element if NULL.

RETURNS A pointer to changed data

STATUS **ok, invalid_id, out_of_memory, incomplete_record**

get_shared_data_type

DESCRIPTION	The **get_shared_data_type** function is used to get the type associated with a shared data element.

SYNTAX

```
serpent_data_types get_shared_data_type(
  /* element_name: in string */
  /* component_name: in string */
);
```

PARAMETERS

element_name	The name of the shared data element.
component_name	The name of the shared data component, or NULL.

RETURNS The type of the shared data element or record component.

STATUS **ok, null_element_name**

get_status

DESCRIPTION	The `get_status` function returns the current system status.
SYNTAX	`isc_status get_status();`
PARAMETERS	None.
RETURNS	The current status.
STATUS	`None`

get_transaction

DESCRIPTION	The `get_transaction` function is used to synchronously retrieve the id for the next completed transaction.
SYNTAX	`transaction_type get_transaction();`
PARAMETERS	None.
RETURNS	The transaction ID for a completed transaction
STATUS	`ok, system_operation_failed`

FUNCTION

get_transaction_no_wait

DESCRIPTION	The `get_transaction` function is used to asynchronously retrieve the id for the next completed transaction.
SYNTAX	`transaction_type get_transaction_no_wait();`
PARAMETERS	None.
RETURNS	The transaction ID for a completed transaction
STATUS	`ok, system_operation_failed, not_available`

incorporate_changes

DESCRIPTION

The **incorporate_changes** procedure is used to incorporate changes into local process memory without destroying unchanged information.

SYNTAX

```
void incorporate_changes(
  /* id : in id_type */
  /* data : in caddr_t */
);
```

PARAMETERS

id	Existing shared data ID
data	Pointer to data with which to incorporate changes.

STATUS

ok, invalid_id

print_status

DESCRIPTION	The `print_status` procedure prints out a user defined error message and the current status.

SYNTAX

```
void print_status(
   /* error_msg : in string */
);
```

PARAMETERS	`error_msg`	User-defined error message.

STATUS	`None`

purge_transaction

DESCRIPTION	The purge transaction procedure is used to purge a received transaction once the contents of the transaction have been examined and acted upon.

SYNTAX

```
void purge_transaction(
  /* transaction : in transaction_type */
);
```

PARAMETERS	`transaction`	.Existing transaction ID.

STATUS `ok, invalid_id, illegal_receiver`

put_shared_data

DESCRIPTION

The **put_shared_data** call is used to put information into shared data.

SYNTAX

```
void put_shared_data(
  /* transaction : in transaction_type */
  /* id : in id_type */
  /* element_name : in string */
  /* component_name : in string */
  /* data : in caddr_t */
);
```

PARAMETERS

transaction	The transaction to which the shared data should be put.
id	Shared data ID.
element_name	The name of the shared data element.
component_name	The name of the shared data component.
data	Shared data.

STATUS

ok, undefined_shared_data_type, null_element_name, invalid_id

remove_shared_data

DESCRIPTION	The **remove_shared_data** procedure is used to remove a specified shared data instance from the shared database.

SYNTAX

```
void remove_shared_data(
  /* transaction : in transaction_type */
  /* element_name : in string */
  /* id : in id_type */
);
```

PARAMETERS

transaction	Transaction from which to remove the shared data element.
element_name	Name of element to be removed.
id	Existing shared data ID.

STATUS `ok, out_of_memory, null_element_name, invalid_id`

rollback_transaction

DESCRIPTION	The `rollback_transaction` procedure is used to abort a given transaction and delete the associated transaction buffer.

SYNTAX	`void rollback_transaction(`
	`  /* transaction : in transaction_type */`
	`);`

PARAMETERS	`transaction` Existing transaction ID.
RETURNS	A handle to a newly-created element
STATUS	`ok, invalid_transaction_handle`

PROCEDURE

serpent_init

DESCRIPTION

The **serpent_init** procedure performs necessary initialization of the interface layer.

SYNTAX

```
void serpent_init(
  /* mailbox : in string */
  /* ill_file : in string */
);
```

PARAMETERS

mailbox	MAIL_BOX constant defined in SADDLE generated include file.
ill_file	ILL_FILE constant defined in SADDLE generated include file.

STATUS

ok, out_of_memory, null_mailbox_name, null_ill_file_name, system_operation_failed

PROCEDURE

serpent_cleanup

DESCRIPTION	The **`serpent_cleanup`** procedure performs necessary cleanup of the interface layer.
SYNTAX	**`void serpent_cleanup();`**
PARAMETERS	None.
STATUS	**`ok`**

start_recording

DESCRIPTION

The **start_recording** procedure enables recording. Once **start_recording** has been called, all transactions and associated data will be saved out to the specified file until the **stop_recording** procedure is invoked.

SYNTAX

```
void start_recording(
  /* file_name : in string */
  /* message : in string */
);
```

PARAMETERS

file_name	File to which to write recording.
message	Recording description.

STATUS

ok, io_failure, already_recording

FUNCTION

start_transaction

DESCRIPTION	The `start_transaction` function is used to define the start of a series of shared data modifications.
SYNTAX	`transaction_type start_transaction();`
PARAMETERS	None.
RETURNS	A unique transaction id
STATUS	`ok, out_of_memory, overflow`

stop_recording

DESCRIPTION	The **stop_recording** procedure causes the current recording to be stopped.
SYNTAX	**void stop_recording();**
PARAMETERS	None.
STATUS	**ok, io_failure, invalid_process_record**

[

4. Ada Language Application Development

This part includes two sections:

- How to Develop an Application in Ada: A step by step specification of the tasks involved in developing a Serpent application in the Ada programming language.
- Serpent Ada Language Interface Reference: A detailed description of the types, constants and routines available for developing Serpent applications in the Ada programming language.

4.1. How to Develop an Application in Ada

The main tasks for developing an application for Serpent require that you define the shared data, add information to shared data, and retrieve information from shared data. There are also two additional tasks which may applied: recording and checking status. Each of these tasks is described in the subsections that follow.

4.1.1. Task 1: Defining the Shared Data

Defining shared data involves two steps:

1. Create the shared data definition file.
2. Run the created file through the SADDLE processor.

The following is a brief description of each of these two steps. The *SEI Serpent SADDLE User's Guide* contains a more complete description of both these steps.

Step 1: Create the shared data definition file The shared data definition file defines the type and structure of application information that may be maintained by the Serpent shared database. The shared data definition is defined in an external ASCII file in SADDLE.

Figure 4-1 is a example of a shared data definition file for the sensor site status application. The content of the shared data definition file is independent of the implementation language used.

The file shown in Figure 4-1 contains definitions for the data shared between the application and the dialogue for the sensor site status application. The three records define the type and structure of the sensor, correlation center, and communication line application objects. Note that these records only contain information to define the actual objects; they do not specify how the information is presented to the end user.

Step 2: Run the created file through the SADDLE processor. Once the shared data has been defined, you can run the file through the SADDLE processor to generate an Ada package specification containing Ada type specifications corresponding to the defined shared data structures. This package may then be "withed" in your Ada application in order to declare local variables of the shared data types. This allows you to directly manipulate

```
sensor_site_status:shared data

sensor:  record
  site_abbr:string[3];
  status:integer;
  site:string[50];
  last_message: string[8];
  rfo:string[50];
  etro:string[8];
end record;

correlation_center:record
  name:string[3];
  status:  integer;
end record;

communication_line:record
  from_sensor:id of sensor;
  to_cc:id of correlation_center;
  etro:string[8];
  status:integer;
end record;

end shared data;
```

Figure 4-1: A shared data definition file

shared data structures in Ada. The Ada package specification generated by running the shared data definition file shown in Figure 4-1 through the SADDLE processor is illustrated in Figure 4-2.

In Figure 4-2, the first two lines in the file provide visibility to various serpent types that are used within the package specification. This is followed by the start of the `sensor_site_status` package specification. Immediately defined within the package specification are two well-known constants: `MAIL_BOX` and `ILL_FILE`. These constants are used in Task 2 to initialize Serpent. The three record definitions correspond to the records defined within the shared data definition file.

4.1.2. Task 2: Adding Information to the Shared Database.

Once you have defined the application shared data, you can begin to develop the application. The code segment from the sensor site status application in Figure 4-3 illustrates the basic operations for adding information to the shared database.

Preliminary Task Steps

In preparation for the task of adding information, you need to complete two preliminary steps:

1. With required packaged specifications.
2. Define local variables.

```
with serpent_type_definitions;
use serpent_type_definitions;

package sensor_site_status is

  MAIL_BOX:  constant string :="SSS_BOX";
  ILL_FILE:  constant string :="SSS.ill";

  type sensor_sdd is record
    site_abbr:           string(1..4);
    status:              integer;
    site:                string (1..51);
    last_message:        string(1..9);
    rfo:                 string(1..51);
    etro:                string(1..9);
  end record;

  type correlation_center_sdd is record
    name:                string(1..4);
    status:              integer;
  end record;

  type communication_line_sdd is record
    from_record:  id_type;  --ID of sensor
    to_cc:        id_type;  --ID of correlation_center
    etro:                string(1..9);
    status:              integer;
  end record;

end sensor_site_status;
```

Figure 4-2: Ada language header file

Step P1: With required packaged specifications. The first step is to "with" the **serpent** package specification and the **sensor_site_status** package specification generated by the SADDLE processor. The **serpent** package specification contains the specification of the data types and calls that you will need to interface with Serpent. The **sensor_site_status** package specification contains the shared data types necessary to define local instances of the shared data elements.

Step P2: Define local variables. The next preliminary step is to define the required local variables. The first variable defined is transaction, which is of `transaction_type`. This variable maintains the handle for a created transaction. The next variables to be defined are `cmc` and `oft`, both of which are of type `correlation_center`. These variables store local instances of the data that is going to be shared across the interface with the Serpent system. The type definition for the `correlation_center` structure was automatically generated by the SADDLE processor during step 2 of Task 1.

```
with serpent;
use serpent;

with sensor_site_status;
use sensor_site_status;

procedure main is

  oft_name:  constant string := "OFT";
  green_status:  constant integer := 0;
  yellow_status:  constant integer := 1;
  red_status:  constant integer := 2;

  transaction: transaction_type;
  cmc_id, oft_id: id_type;
  cmc, oft: correlation_center;

begin

  serpent_init(mail_box, ill_file);

  cmc.name(1..cmc_name'length) := cmc_name;
  cmc.status := green_status;

  oft.name(1..oft_name'length) := oft_name;
  oft.status := green_status;

  transaction:=start_transaction;

  oft_id := add_shared_data(
    transaction,
    "correlation_center",
    "",
    oft'address
  );
  cmc_id := add_shared_data(
    transaction,
    "correlation_center",
    "",
    cmc'address
  );

  commit_transaction(transaction);

  serpent_cleanup;

  return;

end main;
```

Figure 4-3: Basic calls for adding information to the shared database

The two variables that follow, `cmc_id` and `oft_id`, store the ids of the shared data instances created in shared data. It is necessary for the application to maintain this information, since it is the only way to correlate end-user updates with local application information when multiple instances of a single shared data element are used.

Main Task Steps

The main task of adding information to the shared database involves five distinct coding steps:

1. Initialize Serpent.
2. Start a transaction.
3. Add shared data to the interface.
4. Commit the transaction.
5. Clean up.

Step 1: Initialize Serpent. Once the appropriate variables have been declared it is possible to begin describing the logic. The first step is to initialize the Serpent system using the `serpent_init` call and passing the `MAIL_BOX` and `ILL_FILE` constants generated by the SADDLE processor during step 2 of task 1.

Step 2: Start a transaction. Before information can be added to the shared database it is necessary to start a transaction. All additions or modifications to the shared database must be performed as part of a transaction.

Step 3: Add information to the shared database. Once a transaction has been started, you can begin to add information to the shared database as part of this transaction.

Step 4: Commit the transaction. The actual change to shared data does not occur until the transaction is committed. Up to this point it is also possible to roll back the transaction so that none of the changes to shared data occur.

Step 5: Clean up. The `serpent_cleanup` routine must be invoked before exiting the application. It is important that you complete this step, to release all allocated system resources.

4.1.3. Task 3: Retrieving Information from the Shared Database

Once application data exists in the shared database it may be presented to the end-user using one or more of the available technologies. The end-user may in turn make modifications to this data. These modifications are sent back to the application to be updated in the application's local database. It is therefore necessary for the application to retrieve information back from the shared database.

The Serpent interface provides both synchronous and asynchronous calls for getting information back from the shared database. The following code segment from the sensor site status application in Figure 4-4 illustrates the basic calls required to synchronously retrieve data from the interface.

```
procedure get_user_updates is

--
--  Constants.
--
  oft_name: constant string := "OFT";
  green_status: constant integer := 0;
  yellow_status: constant integer := 1;
  red_status: constant integer := 2;
--
--  Retained data.
--
  transaction: transaction_type;

  id: id_type;

  shared_data_element: hash_element;

begin
--
--  Retrieve information from shared database.
--
  transaction := get_transaction;

  id := get_first_changed_element(transaction);

  while id /= null_id loop

    shared_data_element := get_from_hashtable(id_table, id);
    incorporate_changes(
      transaction,
      id,
      shared_data_element'address
    );

    id := get_next_changed_element(transaction);

  end loop;

  purge_transaction(transaction);

  return;

end get_user_updates;
```

Figure 4-4: Basic calls required to retrieve data synchronously

Task Steps

The task of retrieving information from the shared database involves three distinct coding steps:

1. Get a transaction.
2. Update local database.
3. Purge transaction.

Step 1: Get a transaction. The first step in retrieving information from the shared data base is to get a transaction. The `get_transaction` routine waits until a transaction is available and then returns a handle for this transaction. To poll for a new transaction asynchronously, it is possible to call the `get_transaction_no_wait` routine, which will return `not_available` if no transaction is available.

Step 2: Update local database. Transactions can be thought of logically as a list of changed elements. The next call, `get_first_changed_element`, returns the id of the first changed element on the list. This id can then be used to access several types of information about the shared data element.

The application must maintain a correlation between the shared data ids and the actual data items to incorporate changes successfully into its existing local data. For the purposes of this example, it is assumed that this database is maintained as a hashtable indexed by the shared data element id. The purpose of the while loop then is to incorporate all of the changes into this local database or hashtable. A pointer to the shared data element to be updated is retrieved from the hashtable using the `get_from_hashtable` routine and passing id as an index into the hashtable. The `incorporate_changes` call then makes the updates to the local description of the shared data elements and whatever changes were made by the dialogue.

The last call within the loop gets the next changed element from the transaction. The loop repeats until a `null_id` is returned.

Step 3: Purge transaction. After the loop ends the transaction can be purged safely. It is your responsibility to ensure that transactions are purged, since this call releases resources that otherwise could run out.

4.2. Recording Shared Database Transactions

There are two major tasks that need to be performed when using the record/playback feature of Serpent for testing the application or user interface:

1. Recording shared database transactions.
2. Testing the application or user interface.

Since the steps involved in the second task can be performed independent of the implementation language they are described later in the Application and Dialogue Testing part of this guide.

Before testing the application or the dialogue however, you must first record the transactions you would like to use in testing. Figure 4-5 illustrates the basic operations for recording transactions.

```
transaction: transaction_type;

begin
                   :
--
-- Start recording.
--
   start_recording("recording", "test data:  5.7.3");
--
-- Send test data.
--
   transaction := start_transaction;

                   :

   commit_transaction(transaction);

   transaction := start_transaction;

                   :

   commit_transaction(transaction);

   transaction := start_transaction;

                   :

   commit_transaction(transaction);
--
-- Stop recording.
--
   stop_recording;

                   :
end main;
```

Figure 4-5: Recording transactions

Task Steps

There are three distinct coding steps involved in recording shared database transactions:

1. Start recording.
2. Send transactions.
3. Stop recording.

Step 1: Start recording. The first step is to begin recording by calling the **start_recording** routine specifying both the name of the file in which to save the recording and a message to help identify the file.

Step 2: Send transactions. After the call is made you may start sending transactions across the interface. You may send any number of transactions containing any type or amount of data.

Step 3: Stop recording. Once **start_recording** has been called, all transactions and associated data will be saved out to the specified file until the **stop_recording** routine is invoked.

4.2.1. Checking Status

Each routine in Serpent sets status on exit. It is good software engineering practice to check this status after every call to make sure that the routine has executed correctly, and provide appropriate recovery actions if it has not. Figure 4-6 shows the operations that Serpent provides for examining the status.

```
transaction := start transaction;
if get_status /= ok then
  print_status("bad status from start transaction");
  return;
end if;
```

Figure 4-6: Operations for examining the status

The first of these status calls is the **get_status**, which returns an enumeration of status codes. Valid status that each routine in Serpent may return are defined in the reference sections of this developer's guide.

The **print_status** routine prints out a user-defined error message and the current status.

4.3. Serpent Ada Language Interface

4.3.1. Types and Constants

This subsection contains the type and constant definitions that are used in the Ada language interface to the Serpent system. The following is a list and short description of each of these types and constants. A more complete description immediately follows:

Type/Constant	Description
`buffer`	used to define the structure of a shared data buffer
`change_type`	defines the type of modification made for an element
`id_type`	used to uniquely identify shared data elements
`null_id`	defines the null value for the `id_type`
`serpent_data_types`	an enumeration of defined Serpent data types
`transaction_type`	used to define transaction handles
`undefined values`	Constants corresponding to undefined values for all supported types

TYPE

buffer

DESCRIPTION

The **buffer** type is used to define the structure of a buffer within shared data.

DEFINITION

```
type buffer is record {
  length : integer;
  body : system.address;
end record;
```

COMPONENTS

length	Length of the buffer in bytes.
body	Address of the actual buffer data.

change_type

DESCRIPTION The change_type defines the type of modification made for an element.

DEFINITION

```
type change_type is (no_change, create, modify,
remove, get);
```

COMPONENTS

`no_change`	Not changed or invalid change.
`remove`	Remove existing shared data instance.
`create`	Newly created shared data instance.
`modify`	Modified existing shared data instance.
`remove`	Remove existing shared data instance.
`get`	Get value for existing shared data instance.

id_type

DESCRIPTION The id type is used to uniquely identify shared data elements.

DEFINITION

```
type id_type is new int;
```

null_id

DESCRIPTION The null_id constant defines the null value for the id_type. This constant can be used to test for null id values.

DEFINITION

```
null_id : constant id_type;
```

serpent_data_types

DESCRIPTION The serpent data types type is an enumeration of the defined Serpent data types.

DEFINITION

```
type serpent_data_types is (
  serpent_boolean,
  serpent_integer,
  serpent_real,
  serpent_string,
  serpent_record,
  serpent_id,
  serpent_buffer
);
```

transaction_type

DESCRIPTION Variables of transaction_type are used to define transactions.

DEFINITION

```
type transaction_type is private;
```

undefined values

DESCRIPTION

The following constants correspond to undefined values for all types supported by Serpent. These constants can be used to test for ndefined shared data components. When checking for an undefined record value it is best to check the buffer length failed for **UNDEFINED_BUFFER_LENGTH**.

DEFINITION

```
UNDEFINED_INTEGER : constant integer;
UNDEFINED_INTEGER : constant integer;
UNDEFINED_REAL : constant real;
UNDEFINED_STRING : constant string;
UNDEFINED_RECORD : constant record;
UNDEFINED_ID : constant id_type;
UNDEFINED_BUFFER_LENGTH : constant integer;
UNDEFINED_BUFFER_BODY : constant integer;
```

4.3.2. Routines

This subsection describes the routines that make up the C language interface to Serpent. These routines fall into the following categories:

- Initialization/cleanup
  - serpent_init
  - serpent_cleanup
- Transaction processing
  - start_transaction
  - commit_transaction
  - rollback_transaction
  - get_transaction
  - get_transaction_no_wait
  - purge_transaction
  -
- Sending and retrieving data
  - add_shared_data
  - put_shared_data
  - remove_shared_data
  - get_first_changed_element
  - get_next_changed_element
  - get_shared_data
  - incorporate_changes
  - create_changed_component_list
  - destroy_changed_component_list
  - get_change_type
  - get_element_name
  - get_shared_data_type
  -
- Record/playback
  - start_recording
  - stop_recording
- Checking Status
  - get_status
  - print_status

add_shared_data

DESCRIPTION

The **add_shared_data** routine creates an instances for the specified shared data element and returns a unique ID. The shared data instanced may or may not be initialized.

SYNTAX

```
function add_shared_data (
  transaction : in transaction_type;
  element_name, component_name : in string;
  data : in system.address
) return id_type;
```

PARAMETERS

transaction	The transaction for which this operation is defined.
element_name	The name of the shared data element.
component_name	The name of a specific component to be initialized with the data or null if the data corresponds to the entire element.
data	data or null pointer if non-initialized.

RETURNS

The ID of the newly created shared data instance.

STATUS

ok, out_of_memory, null_element_name, overflow)

ROUTINE

commit_transaction

DESCRIPTION	The `commit_transaction` procedure is used to commit a transaction to the shared database.

SYNTAX

```
procedure commit_transaction(
  transaction : in transaction_type
);
```

PARAMETERS	`transaction`	Existing transaction ID.

STATUS	`ok, out_of_memory, invalid_transaction_handle`

create_changed_component_list

DESCRIPTION

The **create_changed_component_list** function accepts an instance id as a parameter and creates a list of changed component names.

SYNTAX

```
function create_changed_component_list(
  id : in id_type
) return LIST;
```

PARAMETERS	**id**	Existing data instance id

RETURNS

The list of changed component names associated with a data instance, or **NULL** if none.

STATUS

ok, invalid_id, out_of_memory, element_not_a_record

destroy_changed_component_list

DESCRIPTION	The **destroy_changed_component_list** procedure releases storage associated with a changed component list.

SYNTAX

```
procedure destroy_changed_component_list(
  changed_component_list : in LIST
);
```

PARAMETERS

changed_component_list
List to be destroyed.

STATUS **ok**

get_change_type

DESCRIPTION	The **get_change_type** function accepts an instance id as a parameter and returns the associated change type.

SYNTAX

```
function get_change_type(
  id : in id_type
) return change_type;
```

PARAMETERS	**id** — Existing shared data ID
RETURNS	Element name associated with the shared data instance ID.
STATUS	**ok, invalid_change_type, invalid_transaction_handle, invalid_id**

get_element_name

DESCRIPTION	The **get_element_name** function accepts an instance id as a parameter and returns the associated element name.

SYNTAX

```
function get_element_name(
  id : in id_type
) return string;
```

PARAMETERS	**id**	Existing shared data ID.

RETURNS	Element name associated with the shared data instance ID
STATUS	**ok, invalid_id**

get_first_changed_element

DESCRIPTION The **get_first_changed_element** function is used to get the id of the first changed element on a transaction list.

SYNTAX

```
function get_first_changed_element(
  transaction : in transaction_type
) return id_type;
```

PARAMETERS

transaction	Existing transaction ID

RETURNS The handle of the first changed element

STATUS **ok, invalid_transaction_handle, out_of_memory**

get_next_changed_element

DESCRIPTION	The **get_next_changed_element** function is used to get the id of the next changed element on a transaction list or return **null_id** if the transaction list is empty.

SYNTAX

```
function get_next_changed_element(
  transaction : in transaction_type
) return id_type;
```

PARAMETERS	**transaction**	Existing transaction ID

RETURNS	The handle of the next changed element
STATUS	**ok, invalid_transaction_handle, out_of_memory**

get_shared_data

DESCRIPTION

The **get_shared_data** function allocates process memory, copies shared data into process memory and returns a pointer to the data.

Warning: Record components may not have been specified and, therefore, would not contain valid data.

SYNTAX

```
function get_shared_data(
  transaction : in transaction_type
  id : in id_type
  component_name : in string
) return system.address;
```

PARAMETERS

transaction	Transaction in which to find the shared data id.
id	Existing shared data id.
component_name	Name of component for which to retrieve data, or entire element if NULL.

RETURNS

A pointer to changed data

STATUS

ok, invalid_id, out_of_memory, incomplete_record

get_shared_data_type

DESCRIPTION	The **get_shared_data_type** function is used to get the type associated with a shared data element.

SYNTAX

```
function get_shared_data_type(
  element_name, component_name : in string
) return serpent_data_types;
```

PARAMETERS

element_name	The name of the shared data element.
component_name	The name of the shared data component, or NULL.

RETURNS The type of the shared data element or record component.

STATUS **ok, null_element_name**

get_status

DESCRIPTION	The **get_status** function returns the current system status.
SYNTAX	`function get_status return status_codes;`
PARAMETERS	None.
RETURNS	The current status.
STATUS	**None**

get_transaction

DESCRIPTION	The `get_transaction` function is used to synchronously retrieve the id for the next completed transaction.
SYNTAX	`function get_transaction return transaction_type;`
PARAMETERS	None.
RETURNS	The transaction ID for a completed transaction
STATUS	`ok, system_operation_failed`

get_transaction_no_wait

DESCRIPTION	The **get_transaction** function is used to asynchronously retrieve the id for the next completed transaction.
[BSYNTAX	**function get_transaction_no_wait return transaction_type;**
PARAMETERS	None.
RETURNS	The transaction ID for a completed transaction
STATUS	**ok, system_operation_failed, not_available**

incorporate_changes

DESCRIPTION	The `incorporate_changes` procedure is used to incorporate changes into local process memory without destroying unchanged information.

SYNTAX

```
procedure incorporate_changes(
  id : in id_type;
  data : in system.address
);
```

PARAMETERS		
	`id`	Existing shared data ID
	`data`	Pointer to data with which to incorporate changes.

STATUS `ok, invalid_id`

print_status

DESCRIPTION	The **print_status** procedure prints out a user defined error message and the current status.

SYNTAX

```
procedure print_status(
  error_msg : in string
);
```

PARAMETERS	**error_msg**	User-defined error message.

STATUS	**None**

purge_transaction

DESCRIPTION	The purge transaction procedure is used to purge a received transaction once the contents of the transaction have been examined and acted upon.
SYNTAX	`procedure purge_transaction( transaction : in transaction_type );`
PARAMETERS	**transaction** Existing transaction ID.
STATUS	**ok, invalid_id, illegal_receiver**

```
procedure purge_transaction(
  transaction : in transaction_type
);
```

put_shared_data

DESCRIPTION	The `put_shared_data` call is used to put information into shared data.

SYNTAX

```
procedure put_shared_data(
  transaction : in transaction_type;
  id : in id_type;
  element_name : in string;
  component_name : in string;
  data : in system.address
);
```

PARAMETERS

`transaction`	The transaction to which the shared data should be put.
`id`	Shared data ID.
`element_name`	The name of the shared data element.
`component_name`	The name of the shared data component.
`data`	Shared data.

STATUS

`ok, undefined_shared_data_type, null_element_name, invalid_id`

remove_shared_data

DESCRIPTION	The **remove_shared_data** procedure is used to remove a specified shared data instance from the shared database.

SYNTAX

```
procedure remove_shared_data(
  transaction : in transaction_type;
  element_name : in string;
  id : in id_type
);
```

PARAMETERS

transaction	Transaction from which to remove the shared data element.
element_name	Name of element to be removed.
id	Existing shared data ID.

STATUS **ok, out_of_memory, null_element_name, invalid_id**

rollback_transaction

DESCRIPTION The `rollback_transaction` procedure is used to abort a given transaction and delete the associated transaction buffer.

SYNTAX

```
procedure rollback_transaction(
  transaction : in transaction_type
);
```

PARAMETERS

`transaction`	Existing transaction ID.

RETURNS A handle to a newly-created element

STATUS `ok, invalid_transaction_handle`

serpent_init

DESCRIPTION

The **`serpent_init`** procedure performs necessary initialization of the interface layer.

SYNTAX

```
procedure serpent_init(mailbox, ill_file : in
string);
```

PARAMETERS

`mailbox`	MAIL_BOX constant defined in SADDLE generated include file.
`ill_file`	ILL_FILE constant defined in SADDLE generated include file.

STATUS

`ok, out_of_memory, null_mailbox_name, null_ill_file_name, system_operation_failed`

serpent_cleanup

DESCRIPTION	The `serpent_cleanup` procedure performs necessary cleanup of the interface layer.
SYNTAX	`procedure serpent_cleanup;`
PARAMETERS	None.
STATUS	`ok`

start_recording

DESCRIPTION

The **start_recording** procedure enables recording. Once **start_recording** has been called, all transactions and associated data will be saved out to the specified file until the **stop_recording** procedure is invoked.

SYNTAX

```
procedure start_recording(
  transaction : in transaction_type;
  file_name : in string
);
```

PARAMETERS

file_name	File to which to write recording.
message	Recording description.

STATUS

ok, io_failure, already_recording

FUNCTION

start_transaction

DESCRIPTION	The **`start_transaction`** function is used to define the start of a series of shared data modifications.
SYNTAX	**`function start_transaction return transaction_type;`**
PARAMETERS	None.
RETURNS	A unique transaction id
STATUS	**`ok, out_of_memory, overflow`**

PROCEDURE

stop_recording

DESCRIPTION	The `stop_recording` procedure causes the current recording to be stopped.
SYNTAX	`procedure stop_recording;`
PARAMETERS	None.
STATUS	`ok, io_failure, invalid_process_record`

5. Application and Dialogue Testing

5.1. Playback/Record

There are two major tasks that need to be performed when using the record/playback feature of Serpent for testing the application or user interface:

1. Recording shared database transactions.
2. Testing the application or user interface.

The steps involved in the specification of the first task are dependent on the language in which the application was developed; therefore a description of the steps involved in recording shared database transactions is included in both the C language and Ada language application development parts of this guide.

5.1.1. Testing the Application

Once you have made a recording it is possible to use that recording to test either the application or the dialogue. This is accomplished by using the recording to simulate the user interface (when testing the application) or the application (when testing the dialogue). In order to test the Sensor Site Status application (sss), for example, you would run the app-test command provided with Serpent specifying both the application to be tested and the name of the file containing the recorded test data, as illustrated in Figure 5-1.

```
% app-test sss recording
Playing back journal file: recording
Message: regression test data, 5.7.3
Playback completed successfully
% _
```

Figure 5-1: Testing the Application

The app-test command will then simulate the dialogue manager. This technique allows the application developer to test the application without the dialogue manager. The application must be tested in the same directory as the recording was made.

5.1.2. Testing the Dialogue

The same recording can also be used to test the user interface. In order to test the Sensor Site Status dialogue (sss.dlg), for example, you would run the dialogue-test command provided with Serpent specifying both the name of the dialogue to be tested and the name of the file containing the recorded test data, as illustrated in Figure 5-2.

The dm-test command will then simulate the application. This technique allows the dialogue specifier to test the user interface without the actual application. It is once again important that the dialogue be tested in the same directory as the recording was made.

```
% dm-test sss.dlg recording
Playing back journal file: recording
Message: regression test data, 5.7.3
Playback completed successfully
% _
```

Figure 5-2: Testing the User Interface

5.1.3. Commands

This subsection contains definitions of some commands provided with Serpent to assist in testing Serpent applications and dialogues. The following is a list and short description of each of these commands. A more complete description immediately follows:

Command	Description
`app-test`	used to test an existing application by simulating Serpent execution
`dialogue-test`	used to test an existing dialogue by simulating the application program.

app-test

DESCRIPTION	The application-test command can be used to test an existing application by simulating Serpent execution. The application-test command requires a recording of the application to be made prior to testing. The application must then be tested in the same directory as the recording was made.	
DEFINITION	`app-test application filename`	
PARAMETERS	`application`	The name of the application being tested. The application is assumed to be in the working directory.
	`filename`	The name of the file containing the recording to be played back.
RETURNS	0	ok
	1	application not found
	2	playback file not found
	3	error during playback

dialogue-test

DESCRIPTION

The dialogue-test command can be used to test an existing dialogue by simulating the application program. The dialogue-test command requires a recording of the application to be made prior to testing. The dialogue must then be tested in the same directory as the recording was made.

DEFINITION

```
dialogue-test dialogue filename
```

PARAMETERS

dialogue	The name of the dialogue being tested. The dialogue is assumed to be in the working directory.
filename	The name of the file containing the recording to be played back.

RETURNS

0	ok
1	dialogue not found
2	playback file not found
3	error during playback

Appendix A: Glossary of Terms

application layer	those components of a software system that implement the "core" application functionality of the system.
dialogue	a specification of the presentation of application information to, and interactions with, the end-user.
dialogue layer	Serpent layer that controls the dialogue between the application and the end-user of the application.
dialogue manager	Serpent component that executes the dialogue.
ID	unique shared data instance identifier.
I/O technologies	existing hardware/software systems that perform some level of generalized interaction with the user.
presentation layer	Serpent layer concerned with low level interaction with the user. This layer consists of the various I/O technologies.
presentation independent	independent of the user interface of the system.
shared database	Application and technology data maintained in Serpent.
shared data definition	a description of the type and structure of data that can be placed in the shared database.
shared data element	any shared data structure that may be instantiated at run-time.
shared data instance	an instance of a shared data element.
transaction	a collection of updates to the shared database that is logically processed at one time.
user interface	those components of a software system that specify the presentation of application information to, and interaction with, the end-user.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-89-UG-6	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-89-TR-12
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS.
11. TITLE (Include Security Classification) SEI SERPENT APPLICATIN DEVELOPER'S GUIDE	

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
63752F	N/A	N/A	N/A

12. PERSONAL AUTHOR(S)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
FINAL	FROM ______ TO ______		

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

DIALOGUE SPECIFICATION, PROTOTYPING, SERPENT, USER INTERFACE, USER INTERFACE MANAGEMENT SYSTEM

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

THIS DOCUMENT DESCRIBES HOW TO DEVELOP APPLICATIONS USING SERPENT. SERPENT IS A USER INTERFACE MANAGEMENT SYSTEM (UIMS) BEING DEVELOPED AT THE SOFTWARE ENGINEERING INSTITUTE (SEI). SERPENT SUPPORTS THE DEVELOPMENT AND IMPLEMENTATION OF THE USER INTERFACE FOR A SYSTEM. IT PROVIDES AN EDITOR TO SPECIFY THE USER INTERFACE AND A RUNTIME SYSTEM THAT COMMUNICATES WITH THE APPLICATION TO DISPLAY DATA TO THE END USER.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] SAME AS RPT. ☐ [✓ DTIC USERS]	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION	
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630	22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS P